

Adding Polymorphic Abstraction to ML*

(Detailed Abstract)

A. J. Kfoury
kfoury@cs.bu.edu
Dept. of Computer Science
Boston University

J. B. Wells
jbw@cs.bu.edu
Dept. of Computer Science
Boston University

March 16, 1994

1 Synopsis

The ML programming language restricts type polymorphism to occur only in the “**let-in**” construct and requires every occurrence of a formal parameter of a function (a λ -abstraction) to have the same type. Milner in 1978 [Mil78] refers to this restriction, which was adopted to help ML achieve automatic type inference, as a serious limitation. We show that this restriction can be relaxed enough to allow universal polymorphic abstraction without losing automatic type inference. In other words, the language may allow occurrences of a formal parameter to have types that are substitution instances of a generic type. This extension is equivalent to the rank-2 fragment of system **F**. We precisely characterize the additional program phrases (λ -terms) that can be typed with this extension and we describe typing anomalies both before and after the extension. We discuss how macros may be used to gain some of the power of rank-3 types without losing automatic type inference. We also discuss user-interface problems in how to inform the programmer of the possible types a program phrase (λ -term) may have.

2 Removing the Monomorphic-Abstraction Restriction

We now present Core-ML, a fragment of ML containing the essential parts for our analysis. Core-ML is a λ -calculus augmented with the **let-in** construct and a set of type inference rules. The usual constants of ML, including the fixpoint operator which introduces recursion, are omitted from the language at both the term and type level. We assume the reader is familiar with some notation for typed λ -calculi. The set \mathcal{T} of Core-ML terms is defined by the grammar $\mathcal{T} ::= \mathcal{V} \mid (\mathcal{T} \mathcal{T}) \mid (\lambda \mathcal{V}. \mathcal{T}) \mid (\mathbf{let} \ \mathcal{V} = \mathcal{T} \ \mathbf{in} \ \mathcal{T})$ where \mathcal{V} is the set of term variables. The term $(\mathbf{let} \ x = M \ \mathbf{in} \ N)$ is operationally equivalent to $((\lambda x. N)M)$ although it is typed more permissively. Letting \mathbb{V} be the set of type variables, the allowed set of types is the set \mathbb{U} defined in Figure 1. The set \mathbb{O} of open (monomorphic) types and the set \mathbb{U} of universal (polymorphic) types, are called “types” and “type-schemes”, respectively, in the ML literature. When τ, τ' are types and $\vec{\alpha} = \{\alpha_1, \dots, \alpha_n\}$ is a set of variables, we write $\forall \vec{\alpha}. \tau \preceq \tau'$ whenever there exist types τ_1, \dots, τ_n such that $\tau' = \tau[\alpha_1 := \tau_1, \dots, \alpha_n := \tau_n]$.

We present the inference rules of Core-ML in Figure 2. Although these are not the standard inference rules for ML, these rules type the same set of terms and differ only in that the final derived type for a term must be monomorphic [CDDK86]. Observe the restrictions in the rules that the types τ and τ' must be quantifier-free and can not be polymorphic. Type polymorphism in ML is only allowed for the bound variable of the **let-in** construct. Previous research has shown that type polymorphism can not be allowed for the bound variable of the **letrec-in** construct (which we have omitted from our Core-ML) without losing automatic type inference [KTU93]. However, we show that type polymorphism in the form of universal types may safely be allowed for the bound variable of ordinary λ -abstractions.

*This work is partly supported by NSF grant CCR-9113196.

$$\begin{aligned}
\mathbb{O} &::= \mathbb{V} \mid (\mathbb{O} \rightarrow \mathbb{O}) \\
\mathbb{U} &::= \mathbb{O} \mid (\forall \mathbb{V}. \mathbb{U}) \\
\mathbb{R}(0) &= \mathbb{O} \\
\mathbb{R}(k+1) &::= \mathbb{R}(k) \mid (\mathbb{R}(k) \rightarrow \mathbb{R}(k+1)) \mid (\forall \mathbb{V}. \mathbb{R}(k+1)) \\
\mathbb{S}' &::= \mathbb{V} \mid (\mathbb{S} \rightarrow \mathbb{S}') \\
\mathbb{S} &::= \mathbb{S}' \mid (\forall \mathbb{V}. \mathbb{S}) \\
\mathbb{S}(k) &= \mathbb{S} \cap \mathbb{R}(k) \\
\mathbb{S}'(k) &= \mathbb{S}' \cap \mathbb{R}(k)
\end{aligned}$$

Figure 1: Sets of Types Used in this Paper.

VAR	$A \vdash x : \sigma$	$A(x) \preceq \sigma, \quad \sigma \in \mathbb{O}$
APP	$\frac{A \vdash M : \tau \rightarrow \tau', \quad A \vdash N : \tau}{A \vdash (M N) : \tau'}$	$\tau, \tau' \in \mathbb{O}$
ABS	$\frac{A \cup \{x : \tau\} \vdash M : \tau'}{A \vdash (\lambda x. M) : \tau \rightarrow \tau'}$	$\tau, \tau' \in \mathbb{O}$
LET	$\frac{A \vdash N : \tau, \quad A \cup \{x \rightarrow \forall \vec{\alpha}. \tau\} \vdash M : \tau'}{A \vdash (\mathbf{let} \ x = N \ \mathbf{in} \ M) : \tau'}$	$\vec{\alpha} = \text{FTV}(\tau) - \text{FTV}(A), \quad \tau, \tau' \in \mathbb{O}$

Figure 2: Inference Rules of Core-ML.

We now precisely describe the extension to allow *universal polymorphic abstraction* (UPA). We call our extended system Core-ML+UPA. To handle our extension we first extend the type language. In Figure 1, $\mathbb{S}(k)$ is the set of “restricted rank- k types”. We require that all types assigned to variables (members of the range of the function A in a sequent $A \vdash M : \tau$) must be restricted rank-1 types (belong to $\mathbb{S}(1)$) and that all types derived for terms (the type τ in a sequent $A \vdash M : \tau$) must be restricted rank-2 types (belong to $\mathbb{S}(2)$). Then we use the inference rules of Figure 3. The side conditions of the rules in the figure on the types τ and τ' are simple consequences of the restrictions just stated and could be omitted from the figure. The depicted system is exactly equivalent to the system Λ_2^* defined in [KW93], which is equivalent in typing power (but not in derived types) to the system Λ_2 , the rank-2 fragment of the second-order λ -calculus (system **F**). Since there is no LET rule, we make the term $(\mathbf{let} \ x = M \ \mathbf{in} \ N)$ be syntactic sugar for $((\lambda x. N)M)$.

Theorem 2.1 (Core-ML+UPA contains Core-ML) *If there is a Core-ML derivation \mathcal{D} ending with the sequent $A \vdash M : \tau$, then there is a Core-ML+UPA derivation \mathcal{D}' ending with the same sequent.*

Theorem 2.2 (Core-ML+UPA Type Inference Decidable) *There is an effective procedure that given a λ -term M , either generates a type τ where there is a valid Core-ML+UPA derivation ending with a sequent $A \vdash M : \tau$ or correctly states that there is no such derivation.*

Theorem 2.1 is easy for the reader to check. Theorem 2.2 is a result of other recent research [KW93]. At this point, it is reasonable to ask whether the restriction can be lifted any further so that types assigned to variables may be rank-2 or higher types. Unfortunately, it is the case that type inference is undecidable for system Λ_k where $k \geq 3$ [KW93], making it seem unlikely that Core-ML+UPA can be extended with more powerful polymorphism.

VAR	$A \vdash x : \forall \vec{\alpha}. \sigma$	$A(x) \sqsubseteq \sigma, \quad \sigma \in \mathbb{O}, \quad \vec{\alpha} \notin \text{FTV}(A)$
APP	$\frac{A \vdash M : \tau \rightarrow \tau', \quad A \vdash N : \tau}{A \vdash (M N) : \forall \vec{\alpha}. \tau'}$	$\tau \in \mathbb{U}, \quad \tau' \in \mathbb{S}'(2), \quad \vec{\alpha} \notin \text{FTV}(A)$
ABS	$\frac{A \cup \{x : \tau\} \vdash M : \tau'}{A \vdash (\lambda x. M) : \forall \vec{\alpha}. (\tau \rightarrow \tau')}$	$\tau \in \mathbb{U}, \quad \tau' \in \mathbb{S}'(2), \quad \vec{\alpha} \notin \text{FTV}(A)$

Figure 3: Inference Rules of Core-ML+UPA.

3 Practical Implications of Universal Polymorphic Abstraction

The most important practical implication of the Core-ML+UPA system is that it types more terms (program phrases) than Core-ML alone. A simple example of a term typable in Core-ML+UPA but not in Core-ML is:

$$(\lambda y. (\lambda x. x x)) (\lambda w. w) (\lambda z. z)$$

The subterm $(\lambda x. x x)$ could never be typed in Core-ML, because x must be assigned a polymorphic type in order for the subterm $(x x)$ to be typed, but then the abstraction would fail because the ABS rule of Core-ML is too weak. The additional terms typable by Core-ML+UPA overcome two typing anomalies of Core-ML. The first anomaly is that although an open term M may be Core-ML-typable, it is not necessarily the case that $(\lambda x. M)$ is Core-ML-typable for $x \in \text{FV}(M)$, as in the example given above for $(\lambda x. x x)$. Second, even though the term $(\mathbf{let} \ x = M \ \mathbf{in} \ N)$ is operationally equivalent to $((\lambda x. N) M)$, it is typed by a different and more permissive rule. In Core-ML+UPA, both of these anomalies disappear: the λ -closure of any typable term is also typable and $(\mathbf{let} \ x = M \ \mathbf{in} \ N)$ becomes mere syntactic sugar rather than an essential construct.

Although Core-ML+UPA has more typing power than Core-ML, it has its own typing anomaly and other potential drawbacks. The new typing anomaly is that although a term M may be Core-ML+UPA-typable, it is not necessarily the case that there is any term N such that $(N M)$ is Core-ML+UPA-typable. For example, there is no term N such that $(N(\lambda x. x x))$ is typable in Core-ML+UPA. This results from the fact that any type derivable for the subterm $(\lambda x. x x)$ is rank-2 but not rank-1. Thus, in order for the application $(N(\lambda x. x x))$ to be typed, the type derived for N must be rank-3 but not rank-2, which is forbidden in Core-ML+UPA. A term like $(\lambda x. x x)$ which requires a rank-2 but not rank-1 type must have a fixed and predetermined argument, although the argument need not be immediately adjacent to it as in a β -redex. One particularly nasty consequence of this limitation is that a term like $(\lambda x. x x)$ may not be bound to a variable as in the ML fragment “**fun** $f \ x = x \ x$ ”. If this were allowed then type inference would become undecidable since this is equivalent to the context $((\lambda f. [\])(\lambda x. x x))$ which requires rank-3 types.

Despite these limitations on the use of rank-2 types, the presence of polymorphic abstraction makes possible the use of macros to simulate some of the power of rank-3 typing. Without polymorphic abstraction, there is little reason to use macros in ML, since whatever polymorphism can be gained using macros can be gained with the **let-in** construct as well. Consider the following example (slightly adjusted) given by Milner in 1978 to illustrate what he calls “the main limitation” of ML typing [Mil78, p. 356]. The ML type checker rejects the term:

$$\mathbf{let} \ m = (\lambda f. \lambda x. \lambda y. c(f x)(f y)) \ \mathbf{in} \ (c(m(\lambda z. z)3 \ \mathbf{true})(m(\lambda z. z)\mathbf{false} \ \mathbf{true}))$$

The problem with the example is that the type assigned to f must be polymorphic as well as the type derived for the abstraction $(\lambda f. \lambda x. \lambda y. c(f x)(f y))$. Not only does this require polymorphic abstraction, which ML does not have, but the type assigned to the variable m must be rank-2, which is beyond even the power of Core-ML+UPA. However, in Core-ML+UPA, there is the practical alternative of using macros that will not

work in Core-ML. The following term is typable in Core-ML+UPA extended with numbers and booleans but not in ML since it involves polymorphic abstraction rather than the **let-in** construct:

$$(\lambda f.\lambda x.\lambda y.c(fx)(fy))(\lambda z.z)3 \text{ true}$$

Thus, Core-ML+UPA extended with a macro substitution facility would be able to type the following short program which is equivalent to Milner's example above:

```
defmacro m = ( $\lambda f.\lambda x.\lambda y.c(fx)(fy)$ )
c(m( $\lambda z.z$ )3 true)(m( $\lambda z.z$ )false true)
```

A serious drawback to the Core-ML+UPA system is the lack of *principal types*. For a term M typable in Core-ML+UPA with a rank-1 type (in $\mathbb{S}(1)$), there is a type $\tau \in \mathbb{S}(1)$ such that for any type τ' derivable for M such that $\tau' \in \mathbb{S}(1)$, it is the case that $\tau \preceq \tau'$. Thus, each term has a most-general rank-1 type. The same does not hold for rank-2 types. For example, for the term $(\lambda x.x)$ both the types $\varphi = (\forall\alpha.\alpha) \rightarrow (\beta \rightarrow \beta)$ and $\psi = (\forall\alpha.\alpha \rightarrow \alpha) \rightarrow (\beta \rightarrow \beta)$ can be derived. However, there is no type τ derivable for $(\lambda x.x)$ such that both $\tau \preceq \varphi$ and $\tau \preceq \psi$. The primary resulting problem is that we do not know how to give the programmer a concise, easy-to-understand representation of the possible types a term may have. Without such a representation, the programmer may not know enough to be able to expect that a certain term will be typable. We intend to research this problem further.

Fortunately, the additional terms typable in Core-ML+UPA but not in Core-ML can be characterized in a precise and simple way. The transformation θ_4 from [KW93] transforms a term of the form $((\lambda x.(\lambda y.N))P)$ to $(\lambda y.((\lambda x.N)P))$ (some restrictions in [KW93] are ignored here). The transformation θ_4 can be treated as a *reduction* relation. It can be easily checked that θ_4 -reduction preserves β -equivalence and terminates quickly. If the term M is typable in Core-ML+UPA, then the θ_4 -normal form of M looks like $(\lambda x_1 \dots \lambda x_n.Q)$ where $n \geq 0$ and Q is not an abstraction and it is the case that the term Q is typable in Core-ML. If the term M is typable in Core-ML+UPA with a final derived type that is rank-1, then the θ_4 -normal form of M is typable in Core-ML. An interesting consequence of the facts stated here is that rather than extending the type system of ML to handle universal polymorphic abstraction, an equivalent approach would be to have the ML compiler convert all terms to θ_4 -normal form before inferring types for them.

References

- [CDDK86] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *Proc. ACM Conf. LISP Funct. Program.*, pp. 13–27, 1986.
- [DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Conf. Rec. 9th Ann. ACM Symp. Princ. Program. Lang.*, pp. 207–212, 1982.
- [KTU93] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Trans. Prog. Lang. Syst.*, 15(2):290–311, Apr. 1993.
- [KW93] A. J. Kfoury and J. B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order λ -calculus. Tech. Rep. 93-017, Comput. Sci. Dept., Boston Univ., 1993.
- [Mil78] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17:348–375, 1978.